# Ch 11 : Search Trees

## 11.1 Binary Search Trees

We use a search-tree structure to efficiently implement a sorted map

***Binary search tree***: binary tree storing keys (or key-element pairs\_ at its internal nodes and satisfying the following property:

* Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have key(u) ≤ key(v) ≤ key(w)

External nodes do not store items

Inorder traversal of a binary search trees visits the keys in increasing order

**proper binary tree/ full binary tree**: tree in which every node other than the leaves has 2 children

***complete binary tree***: tree in which every level (except last) is completely filled and all nodes are far left possible.

|  |  |  |
| --- | --- | --- |
| ***proper but not complete*** | ***complete but not proper*** | ***proper and complete*** |
| Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 3.19.28 PM.png | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 3.19.44 PM.png | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 3.19.58 PM.png |

### Search

|  |  |
| --- | --- |
| * To search for a key k, we trace a downward path starting at the root * The next node visited depends on the comparison of k with the key of the current node * If we reach a leaf, the key is not found * Example: get(4)   Call TreeSearch(4,root)   * The algorithms for floorEntry and ceilingEntry are similar | **Algorithm *TreeSearch***(***k***, ***v***)  **if *T.isExternal*** (***v***)   **return *null* if *k*** < ***key***(***v***)  **return *TreeSearch***(***k***, ***T.left***(***v***))  **else if *k*** = ***key***(***v***)   **return *v* else** { ***k*** > ***key***(***v***) }   **return *TreeSearch***(***k***, ***T.right***(***v***))  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 2.59.31 PM.png |

### Insertion

|  |  |
| --- | --- |
| * To perform operation **put**(k, o), we search for key *k* (using TreeSearch) * Assume *k* is not already in the tree, and let *w* be the leaf reached by the search * We insert *k* at node *w* and expand *w* into an internal node * Example: insert 5 | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 3.01.23 PM.png |

Deletion

|  |  |
| --- | --- |
| * To perform **remove**(k), we search for key *k* * Assume *k* is in the tree, and let *v* be the node storing *k* * If node *v* has a leaf child *w*, we remove *v* and *w* from the tree with operation **removeExternal**(*w*), which removes *w* and its parent * Example: remove 4   We consider the case where the key *k* to be removed is stored at a node *v* whose children are both internal.   * we find the internal node *w* that follows *v* in an inorder traversal * we copy *key*(*w*) into node *v* * we remove node *w* and its left child *z* (which must be a leaf) by means of operation **removeExternal**(*z*)   Example: remove 3 | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 3.03.45 PM.png  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 3.03.54 PM.png |

### Performance

Consider a multimap with *n* items implemented by means of a binary search tree of height *h*

* space used is ***O(n)***
* methods get, floorEntry, ceilingEntry, put and remove take ***O(h)*** time

Height *h* is ***O(n)*** in ***worst case*** and ***O(log n)*** in ***best case***

## 11.3 AVL Trees

|  |  |
| --- | --- |
| AVL trees are ***balanced***.  AVL tree is a ***binary search tree*** such that for every internal node *v* of T, the ***heights of the children*** of *v* can differ at most ***1***. | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 3.22.12 PM.png |

### Height of AVL Tree

**Claim**: height of AVL tree storing n keys is O(log n)

**Proof**: let us bound n(h): min number of internal nodes of AVL tree of height h.

* can see that n(1) = 1 and n(2) = 2
* for n>2, AVL tree height h contains root node, one AVL tree of height n-1 and another of height n-2
* n(h) = 1 + n(h-1) + n(h-2)

We are trying to find an lower bound for n(h)

* we know n(h-1) > n(h-2)
* n(h) = 1 + n(h-1) + n(h-2) > n(h-2) + n(h-2) = 2 n(h-2) for h=>3

Now we derive recurrence inequality

* n(h) > 2 n(h-2) > 4 n(h-4) > … > 2i n(h-2i) with h-2i =>1

Since n(2) = 2 -> h-2i = 2 <-> h-2 = 2i <-> i = h/2-1

We solve base case with i = h/2-1

* 2i n(h-2i) = 2h/2-1n(2) = 2h/2-12> 2h/2-1

We derive n(h) > 2h/2-1

* log n(h) > log 2h/2-1 <-> log n(h) > h/2-1 <-> h < 2logn(h)+2

Height of AVL tree is O(log n)

### Insertion

Insertion is as in a BST; always done by expanding external node

Example: insert 54

|  |  |
| --- | --- |
| ***before insertion*** | ***after insertion, unbalance*** |
| Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 3.34.25 PM.png | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 3.34.30 PM.png |
| ***inorder traversal unchanged after restructuring*** | |
| Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 3.36.37 PM.png | |

### Trinode restructuring

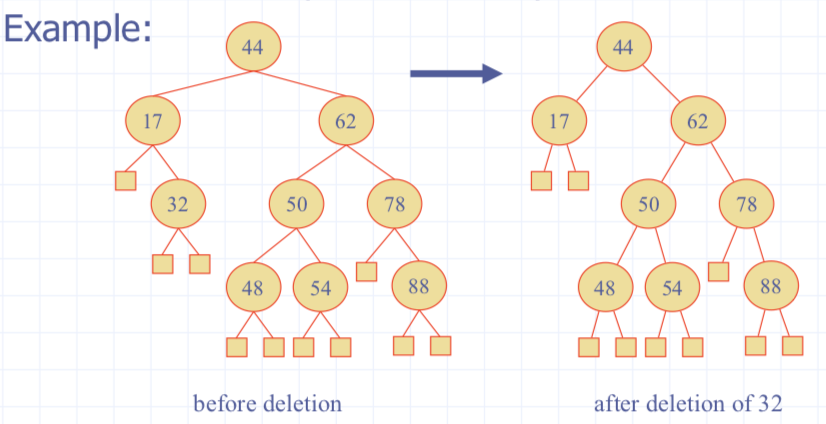
let (a,b,c) be an inorder listing of x, y, z.

perform the rotations needed to make b the topmost node of the tree

|  |  |
| --- | --- |
| ***case 1: single rotation*** (a left rotation about a) | ***case 2: double rotation*** (right about c, then left about a) |
|  |  |
| **Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 3.44.38 PM.png** | **Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 3.44.51 PM.png** |
|  |  |

|  |  |
| --- | --- |
| ***Single rotation*** | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 3.46.19 PM.png |
| ***Double rotation*** | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 3.46.25 PM.png |

### Removal

Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.

### Rebalancing after removal

Let z be the first unbalanced node encountered while traveling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height.  
We perform restructure(x) to restore balance at z.

As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached .

|  |  |
| --- | --- |
| Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 3.48.30 PM.png | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 3.48.43 PM.png |

### Performance

a single restructuring is O(1)

* using a linked-structure binary tree

**get** is O(log n)

* height of tree is O(log n), no restructuring needed

**put** is O(log n)

* initial find is O(log n)
* One Restructuring up the tree, maintaining heights is O(log n)

**remove** is O(log n)

* initial find is O(log n)
* Restructuring up the tree, maintaining heights is O(log n)

